Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные среды и системное программирование

ОТЧЕТ

к лабораторной работе №8

на тему

**ИНТЕРФЕЙС СОКЕТОВ И ОСНОВЫ СЕТЕВОГО ПРОГРАММИРОВАНИЯ (WINDOWS).**

**ПРОГРАММИРОВАНИЕ ВЗАИМОДЕЙСТВИЯ ЧЕРЕЗ**

**СЕТЬ С ИСПОЛЬЗОВАНИЕМ ИНТЕРФЕЙСА**

**СОКЕТОВ. РЕАЛИЗАЦИЯ СЕТЕВЫХ ПРОТОКОЛОВ: СОБСТВЕННЫХ ИЛИ СТАНДАРТНЫХ**

Студент Е. С. Кахновский

Преподаватель Н. Ю. Гриценко

Минск 2023

СОДЕРЖАНИЕ

[1 Цель работы 3](#_Toc146752068)

[2 Теоретические сведения 4](#_Toc146752069)

[3 Результат выполнения 5](#_Toc146752070)

[Заключение 7](#_Toc146752071)

[Список использованных источников 8](#_Toc146752072)

[Приложение А (обязательное) Листинг кода 9](#_Toc146752073)

1. **ЦЕЛЬ РАБОТЫ**

Целью данной лабораторной работы является изучение интерфейса сокетов и основы сетевого программирования (*Windows*). Изучение программирования взаимодействия через сеть с использованием интерфейса сокетов и реализации сетевых протоколов.

1. **ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

Основные сведение:

1. *Windows Sockets* (*WinSock*) представляет собой *API* (интерфейс прикладного программирования) для сетевого программирования в операционной системе *Windows*. Функциональность: *WinSock* обеспечивает приложения возможность создания сокетов (сетевых точек) для обмена данными между компьютерами в сети [1].
2. Сокет представляет собой конечную точку для обмена данными в сетевом взаимодействии. Он характеризуется *IP*-адресом и номером порта. Сокеты используются для установления соединений между клиентами и серверами, а также для передачи данных между ними.
3. Протокол *TCP/IP* (*Transmission Control Protocol/Internet Protocol*) является набором протоколов для передачи данных в сетях, таких как интернет*. TCP/IP* обеспечивает надежную и устойчивую передачу данных между устройствами, подключенными к сети [2].
4. *TCP (Transmission Control Protocol)* – обеспечивает устойчивое, упорядоченное и надежное соединение между двумя точками в сети. Применяется в ситуациях, где важна доставка данных без потерь и в правильном порядке.
5. *UDP (User Datagram Protocol) -* обеспечивает простую и неупорядоченную передачу данных. Применяется в сценариях, где допускаются потери данных, но критически важна минимальная задержка [3]
6. Установление соединения (*Server and Client*): Сервер использует *bind*() для привязки сокета к определенному адресу и порту, затем *listen*() для ожидания входящих соединений. После принятия соединения используется *accept*() для установления связи с клиентом. Клиент использует *connect*() для соединения с сервером по заданному адресу и порту.
7. Для отправки данных сервер использует *send*() или *sendto*(), а клиент - *recv*() или *recvfrom*(). Данные передаются в виде байтового потока. По завершении работы с сокетами важно освободить ресурсы. Для этого используется функция *closesocket*() для закрытия сокета, и *WSACleanup*() для завершения работы с библиотекой *WinSock*.
8. **РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ**

В результате работы было создано два приложения: сервер и клиент.

Реализация серверного приложения (рисунок 1). Создан серверный сокет, привязанный к определенному порту, ожидающий подключений от клиентов. Сокеты клиентов сохраняются для последующей рассылки сообщений. Реализован многопоточный механизм обработки сообщений клиентов.

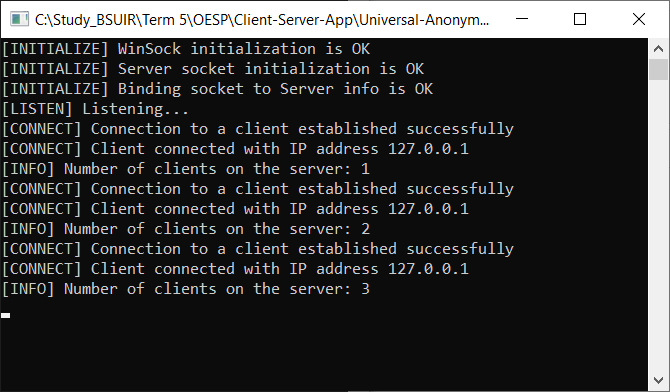


Рисунок 1 – Консольное окно сервера

Создано клиентское консольное окно, позволяющее вводить пользовательское сообщение (рисунок 2), которое в последствии будет отправлено на сервер и продублировано всем остальным клиентам подключившимся к данному серверу. Реализован многопоточный механизм для асинхронного приема сообщений от сервера.

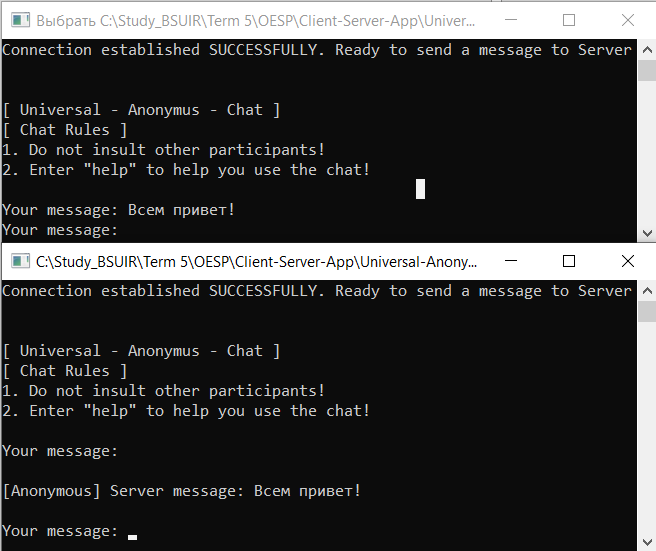


Рисунок 2 – Два запущенных консольных клиента

Так же реализована возможность ввода закодированных команд для получения дополнительной информации *help* (рисунок 3) или выхода из программы *exit*.

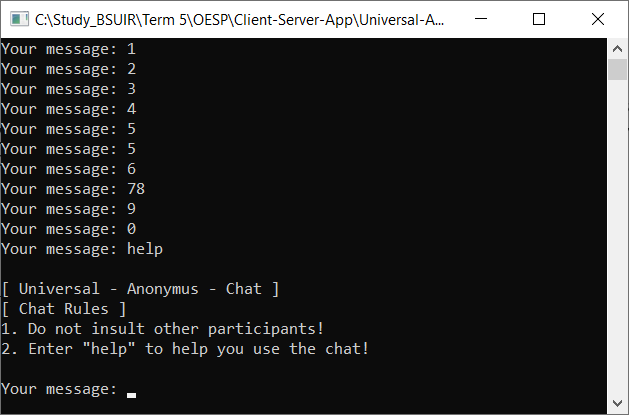


Рисунок 3 – демонстрация команды *help*

# ЗАКЛЮЧЕНИЕ

В ходе лабораторной работы были изучены интерфейс сокетов и основы сетевого программирования (*Windows*). Изучено программирование взаимодействия через сеть с использованием интерфейса сокетов и реализации сетевых протоколов. Для демонстрации данного интерфейса было реализовано два консольных приложения, представляющих собой клиент и сервер.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Локальная вычислительная сеть [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Локальная\_вычислительная\_сеть.
2. Сетевая модель TCP/IP [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/TCP/IP
3. Основы компьютерных сетей [Электронный ресурс]. – Режим доступа: https://habr.com/ru/articles/307252

# ПРИЛОЖЕНИЕ А (обязательное) Листинг кода

Листинг 1 – *Server.cpp*

#include <iostream>

#include <vector>

#include <thread>

#include <mutex>

#include <WinSock2.h>

#include <WS2tcpip.h>

#pragma comment(lib, "Ws2\_32.lib")

using namespace std;

// Глобальный мьютекс для безопасного доступа к вектору сокетов клиентов

std::mutex clientsMutex;

// Функция для обработки сообщений от клиента в отдельном потоке

void HandleClient(SOCKET clientSocket, std::vector<std::shared\_ptr<SOCKET>>& clientSockets)

{

char buffer[1024];

int bytesReceived;

while (true)

{

// Принимаем сообщение от клиента

bytesReceived = recv(clientSocket, buffer, sizeof(buffer), 0);

if (bytesReceived <= 0)

{

// Ошибка или клиент отключился

closesocket(clientSocket);

// Удаляем сокет из вектора

std::lock\_guard<std::mutex> lock(clientsMutex);

auto it = std::find\_if(clientSockets.begin(), clientSockets.end(),

[clientSocket](const std::shared\_ptr<SOCKET>& ptr) {

return \*ptr == clientSocket;

});

if (it != clientSockets.end())

{

clientSockets.erase(it);

}

cout << "[DISCONNECT] Client disconnected " << endl;

cout << "[INFO] Number of clients on the server: " << clientSockets.size() << endl;

break;

}

// Отправляем сообщение всем подключенным клиентам

// std::lock\_guard<std::mutex> lock(clientsMutex);

for (const auto& socketPtr : clientSockets)

{

SOCKET otherClientSocket = \*socketPtr;

if (otherClientSocket != clientSocket)

{

send(otherClientSocket, buffer, bytesReceived, 0);

}

}

}

}

int main()

{

const char IP\_SERV[] = "127.0.0.1";

const int PORT\_NUM = 17177;

const short BUFF\_SIZE = 1024;

int erStat;

in\_addr ip\_to\_num;

erStat = inet\_pton(AF\_INET, IP\_SERV, &ip\_to\_num);

if (erStat <= 0)

{

cout << "[ERROR] Error in IP translation to special numeric format" << endl;

return 1;

}

WSADATA wsData;

erStat = WSAStartup(MAKEWORD(2, 2), &wsData);

if (erStat != 0)

{

cout << "[ERROR] Error WinSock version initialization #" << WSAGetLastError() << endl;

return 1;

}

else

cout << "[INITIALIZE] WinSock initialization is OK" << endl;

SOCKET ServSock = socket(AF\_INET, SOCK\_STREAM, 0);

if (ServSock == INVALID\_SOCKET)

{

cout << "[ERROR] Error initialization socket # " << WSAGetLastError() << endl;

closesocket(ServSock);

WSACleanup();

return 1;

}

else

cout << "[INITIALIZE] Server socket initialization is OK" << endl;

sockaddr\_in servInfo;

ZeroMemory(&servInfo, sizeof(servInfo));

servInfo.sin\_family = AF\_INET;

servInfo.sin\_addr = ip\_to\_num;

servInfo.sin\_port = htons(PORT\_NUM);

erStat = bind(ServSock, (sockaddr\*)&servInfo, sizeof(servInfo));

if (erStat != 0)

{

cout << "[ERROR] Error Socket binding to server info. Error # " << WSAGetLastError() << endl;

closesocket(ServSock);

WSACleanup();

return 1;

}

else

cout << "[INITIALIZE] Binding socket to Server info is OK" << endl;

erStat = listen(ServSock, SOMAXCONN);

if (erStat != 0)

{

cout << "[ERROR] Can't start to listen to. Error # " << WSAGetLastError() << endl;

closesocket(ServSock);

WSACleanup();

return 1;

}

else

{

cout << "[LISTEN] Listening..." << endl;

}

// Вектор для хранения сокетов клиентов

std::vector<std::shared\_ptr<SOCKET>> clientSockets;

while (true)

{

sockaddr\_in clientInfo;

int clientInfo\_size = sizeof(clientInfo);

ZeroMemory(&clientInfo, clientInfo\_size);

SOCKET ClientConn = accept(ServSock, (sockaddr\*)&clientInfo, &clientInfo\_size);

if (ClientConn == INVALID\_SOCKET)

{

cout << "[ERROR] Client detected, but can't connect to a client. Error # " << WSAGetLastError() << endl;

closesocket(ServSock);

closesocket(ClientConn);

WSACleanup();

return 1;

}

else

{

cout << "[CONNECT] Connection to a client established successfully" << endl;

char clientIP[22];

inet\_ntop(AF\_INET, &clientInfo.sin\_addr, clientIP, INET\_ADDRSTRLEN);

cout << "[CONNECT] Client connected with IP address " << clientIP << endl;

// Добавляем сокет клиента в вектор

std::shared\_ptr<SOCKET> clientSocketPtr = std::make\_shared<SOCKET>(ClientConn);

std::lock\_guard<std::mutex> lock(clientsMutex);

clientSockets.push\_back(clientSocketPtr);

cout << "[INFO] Number of clients on the server: " << clientSockets.size() << endl;

// Создаем поток для обработки сообщений клиента

std::thread clientThread(HandleClient, ClientConn, std::ref(clientSockets));

// Детачим поток, чтобы он завершился самостоятельно

clientThread.detach();

}

}

closesocket(ServSock);

WSACleanup();

return 0;

}

Листинг 2 – *Client.cpp*

#include <iostream>

#include <WinSock2.h>

#include <WS2tcpip.h>

#include <inaddr.h>

#include <stdio.h>

#include <vector>

#include <thread>

#pragma comment(lib, "ws2\_32.lib")

using namespace std;

// Struct for Client Data

struct ClientData {

string name;

int age;

};

// Info about App

void Info()

{

cout << endl << "[ Universal - Anonymus - Chat ]" << endl;

cout << "[ Chat Rules ]" << endl;

cout << "1. Do not insult other participants! " << endl;

cout << "2. Enter \"help\" to help you use the chat! " << endl << endl;

}

void SendMessagesAsync(SOCKET clientSocket) {

vector<char> clientBuff(1024);

while (true) {

cout << "Your message: ";

fgets(clientBuff.data(), clientBuff.size(), stdin);

// Check whether the client wants to stop chatting (exit)

if (clientBuff[0] == 'e'

&& clientBuff[1] == 'x'

&& clientBuff[2] == 'i'

&& clientBuff[3] == 't') {

shutdown(clientSocket, SD\_BOTH);

closesocket(clientSocket);

WSACleanup();

return;

}

// Call help list (help)

if (clientBuff[0] == 'h'

&& clientBuff[1] == 'e'

&& clientBuff[2] == 'l'

&& clientBuff[3] == 'p') {

Info();

continue;

}

// Send the message to the server

int packet\_size = send(clientSocket, clientBuff.data(), clientBuff.size(), 0);

if (packet\_size == SOCKET\_ERROR) {

cout << "Can't send message to Server. Error # " << WSAGetLastError() << endl;

closesocket(clientSocket);

WSACleanup();

return;

}

}

}

int main(void) {

const char SERVER\_IP[] = "127.0.0.1"; // Enter IPv4 address of Server

const short SERVER\_PORT\_NUM = 17177; // Enter Listening port on Server side

const short BUFF\_SIZE = 1024; // Maximum size of buffer for exchange info between server and client

int erStat; // For checking errors in sockets functions

//IP in string format to numeric format for socket functions. Data is in "ip\_to\_num"

in\_addr ip\_to\_num;

inet\_pton(AF\_INET, SERVER\_IP, &ip\_to\_num);

// WinSock initialization

WSADATA wsData;

erStat = WSAStartup(MAKEWORD(2, 2), &wsData);

if (erStat != 0) {

cout << "Error WinSock version initializaion #";

cout << WSAGetLastError();

return 1;

}

else

cout << "WinSock initialization is OK" << endl;

// Socket initialization

SOCKET ClientSock = socket(AF\_INET, SOCK\_STREAM, 0);

if (ClientSock == INVALID\_SOCKET) {

cout << "Error initialization socket # " << WSAGetLastError() << endl;

closesocket(ClientSock);

WSACleanup();

}

else

cout << "Client socket initialization is OK" << endl;

// Establishing a connection to Server

sockaddr\_in servInfo;

ZeroMemory(&servInfo, sizeof(servInfo));

servInfo.sin\_family = AF\_INET;

servInfo.sin\_addr = ip\_to\_num;

servInfo.sin\_port = htons(SERVER\_PORT\_NUM);

erStat = connect(ClientSock, (sockaddr\*)&servInfo, sizeof(servInfo));

if (erStat != 0) {

cout << "Connection to Server is FAILED. Error # " << WSAGetLastError() << endl;

closesocket(ClientSock);

WSACleanup();

return 1;

}

else

cout << "Connection established SUCCESSFULLY. Ready to send a message to Server" << endl;

Info(); // Info About App

//Exchange text data between Server and Client. Disconnection if a Client send "exit"

vector <char> servBuff(BUFF\_SIZE), clientBuff(BUFF\_SIZE); // Buffers for sending and receiving data

short packet\_size = 0;

// Create thread for async sending message

std::thread sendThread(SendMessagesAsync, ClientSock);

while (true) {

// Receive messages from the server

}

// Wait complete sendThread

sendThread.join();

closesocket(ClientSock);

WSACleanup();

return 0;

}